(4)

An Overview of Lynx

Michael L. Scott

Technical Report 308
August 1989

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR 308	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): An Overview of Lynx	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Michael L. Scott	8. CONTRACT OR GRANT NUMBER(s): DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Department, 734 Computer Studies Bldg., University of Rochester, Rochester, NY 14627	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: D. Adv. Res. Proj. Agency, 1400 Wilson Blvd, Arlington, VA 22209	12. REPORT DATE: August 1989
	13. NUMBER OF PAGES: 27 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): U.S. Army, ETL, Fort Belvoir, VA 22060	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

distributed computing, message passing, communication performance, parallel programming languages, remote procedure calls

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A programming language can provide much better support for interprocess communication than a library package can. Most message-passing languages limit this support to communication between the pieces of a single program, but this need not be the case. Lynx facilitates convenient, typesafe message passing not only within applications, but also between applications, and among distributed collections of servers. Specifically, it addresses issues of compiler statelessness, late binding, and protection that allow runtime interaction between processes that were developed independently and that do

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Continued)

not trust each other. Implementation experience with Lynx has yielded important insights into the relationship between distributed operating systems and language run-time support packages, and into the inherent costs of high-level message-passing semantics.

An Overview of Lynx

Michael L. Scott

University of Rochester
Department of Computer Science
Rochester, NY 14627

August 1989

Abstract

A programming language can provide much better support for interprocess communication than a library package can. Most message-passing languages limit this support to communication between the pieces of a single program, but this need not be the case. Lynx facilitates convenient, typesafe message passing not only within applications, but also between applications, and among distributed collections of servers. Specifically, it addresses issues of compiler statelessness, late binding, and protection that allow run-time interaction between processes that were developed independently and that do not trust each other. Implementation experience with Lynx has yielded important insights into the relationship between distributed operating systems and language run-time support packages, and into the inherent costs of high-level message-passing semantics.

1. Introduction

A programming language has clear advantages over a library package for communication between processes in a distributed environment.[1] These advantages include non-procedural syntax, direct use of program variables and types, type checking, exception handling, and support for concurrent conversations. Unfortunately, most existing distributed languages are better suited to communication between the processes of a single application than they are to communication between processes that are developed independently. Such independent development is characteristic both of the systems software for multicomputers and of the applications software for geographically-distributed networks. Lynx [19,21] is a message-passing language designed to support both application and system software in a single conceptual framework. It extends the advantages of language-based communication to processes designed in isolation, and compiled and placed in operation without knowledge of their peers.

Lynx was developed at the University of Wisconsin, where it was first implemented on the Charlotte multicomputer operating system [2,11]. Charlotte was designed without Lynx, but experience with a conventional library interface to the kernel suggested that language support for communication could make the

[1] Throughout this paper, the term "process" is used to denote a heavyweight entity with its own address space, supported by the operating system. Active entities in the same address space will be called lightweight threads of control.

programmer's life much easier. Particularly troublesome was the construction of operating system server processes, which needed to communicate conveniently, safely, and efficiently with an ever-changing pool of clients, most of whom could be expected to have been written long after the server was placed in operation. Lynx was designed to provide an unusually high degree of support for dynamic changes in interconnection topology, protection from untrusted processes, and concurrent, interleaved conversations with an arbitrary number of communication partners. To permit the construction of servers, Lynx was designed to provide these benefits without depending on any sort of global information at compile time.

The Lynx compiler is a pure translator: it requires no input other than a source program and produces no output other than an object program. In particular, it does not depend on a database of type definitions or interface descriptions in order to guarantee the validity of messages. A novel application of hashing [22] provides efficient run-time type checking on messages at negligible cost and at a very high level of confidence.

Within the context of independent compilation, Lynx supports reconfiguration and protection with a virtual-circuit abstraction called the *link*. It maintains context for multiple conversations by combining message passing with the scheduling of lightweight threads of control. A link is a symmetric two-directional channel, like a pair of tin cans connected by string. The cans themselves (link ends) are first-class objects that can be created, destroyed, stored in data structures, and passed in messages. It is by passing them in messages that the process connection graph is changed. Threads are a program structuring tool that allows a sequential execution path to be associated with each logically separate conversation. A file server, for example might have a separate thread of control for each of its open files. Each thread could then use straight-line code to perform operations on behalf of a client. Special operations (seeks, for example) could be performed by nested threads that share file-specific data structures. Lynx allows these threads to be created automatically, in response to incoming requests.

In addition to systems software, Lynx has been used to implement a number of parallel and distributed applications. At the University of Rochester, the compiler has been ported to the BBN Butterfly multiprocessor and its Chrysalis operating system. Experience with the two implementations, together with paper designs for two others, has led to important insights into the relationship between a language run-time package and the underlying operating system [19]. A detailed performance study of the Chrysalis implementation has helped to provide a deeper understanding of the inherent costs of message-passing systems [21].

2. Motivation

2.1. Message-Passing Languages

An operating system that allows processes in separate address spaces to communicate outside the file system will generally provide kernel calls for message passing. User programs will access these calls through a traditional subroutine library. Experience with a wide variety of message-passing systems, however, suggests that users find this sort of traditional interface inadequate. The crux of the problem is that interprocess communication is significantly more complicated, from the user's point of view, than are other kernel services.

Structured Information

Like file system read and write operations, library-based communication primitives generally transfer uninterpreted streams of bytes. The desire to impose structure on those bytes has often been a motivation for language-level file system interfaces, and that motivation applies even more strongly to messages. Programmers want to be able to send and receive program variables by name, including those with structured and abstract types, without sacrificing type checking and without explicitly packing and unpacking buffers.

Error Handling and Protection

Interprocess communication is more error-prone than other kernel operations. Both hardware and software may fail. Software is a particular problem, since processes that are not part of the same application cannot in general trust each other. Compared to an open file, a connection to an arbitrary process can display vastly less predictable behavior. Fault-tolerant algorithms may allow a process to recover from many kinds of detectable failures, but it can be awkward to deal with those errors in line by examining kernel call return codes.

Concurrent Conversations

While a conventional sequential program rarely has anything interesting to do while waiting for a kernel call to complete, a process in a distributed environment is much more likely to be budgeting its time among multiple activities. A server, for example, may be working on behalf of multiple clients at once. It cannot afford to be blocked while waiting for a particular client to respond. The kernel can help by providing non-blocking sends and receives, but then the server begins to resemble a state machine more than it resembles straight-line code. The inevitable interleaving of separate conversations leads to very obscure programs.

A language-based approach to communication can improve the situation significantly.

(1) Communication statements in a language can make direct use of program variables and eliminate the need to think about message buffers. The compiler can generate code to *gather* variables to be sent into a buffer acceptable to the operating system. At the message's destination the compiler can arrange to *scatter* the contents of the buffer into appropriate variables in the receiving process. Since the compiler knows the types of all the values involved, it can take steps to ensure that the sender and receiver agree on an interpretation for the bytes contained in the message.

(2) If the programming language includes facilities for exception handling, many of the errors that may occur during communication can be passed to exception handlers outside the normal flow of control. A single exception handler can protect a large number of communication statements. Certain kinds of errors may even be handled in the run-time support package, without ever becoming visible to the programmer.

(3) If the programming language includes facilities for concurrency, separate conversations may be handled by separate threads of control. The run-time support package can be designed to include a *dispatcher* routine that examines each incoming message and makes it available to the appropriate thread.

Each of these numbered points corresponds directly to one of the problems with library-based communication. In addition, a language-based approach to communication can offer specialized syntax and

can arrange for useful side-effects to communication statements. It would be difficult, for example, to provide the functionality of an Ada *select* statement [27] without its distinctive syntax. A less widely appreciated feature of Ada is its carefully-designed semantics for data sharing between tasks. The language reference manual requires a "shared" variable to have a single, consistent value only at the times when tasks exchange messages. An Ada implementation can choose to replicate variables at multiple sites and can allow the copies to acquire inconsistent values, so long as it reconciles the differences before the programmer can detect them — i.e. before the variables can be compared to a value in a message. In a similar vein, the compiler for NIL [24] tracks the status of every program variable and treats a variable that has just been sent in a message as if it were uninitialized. This facility allows the run-time system to implement message passing on a shared-memory machine by moving pointers, without worrying that a sender will subsequently modify the variables it has "sent." In Argus [15], one can send messages between processes that use completely different implementations of a common abstract type. The compiler inserts code in the sender to translate data into a universal, machine-independent format. It inserts code in the receiver to translate that data back into an appropriate local version of the abstraction.

The advantages of language-level communication can be realized either by designing a language from scratch or by augmenting an existing language. The latter approach is typified by remote procedure call *stub* generators. A stub generator is a program that translates a set of declarations for remote operations into ordinary procedures, which can be called from a conventional language, and which hide the details of the underlying message passing. Though a stub generator cannot provide non-procedural syntax or fancy side effects, it can eliminate most of the immediate problems with straightforward use of a kernel call library. By processing the user's declarations it can provide type checking and automatic scatter/gather of message parameters. In conjunction with a language that includes exception handling, it can eliminate most of the complexity of examining kernel call return codes. If the language includes concurrency, it may even be possible to write a dispatcher that automatically routes incoming messages to the threads of control that want them.[2] One of the most successful stub generators can be found in the Cedar environment at Xerox PARC [3].

2.2. Inter-Program Communication

The bulk of the distributed language literature focuses on the needs of distributed programs — collections of processes that are designed to work together and that constitute the pieces of a single, coherent whole. There are equally important scenarios, however, in which communication must occur between processes that were developed independently. Distributed systems software provides one class of examples. Large-scale distributed applications provide another.

The past decade and a half has seen the development of a large number of distributed operating systems. The typical goal of such systems is make a collection of locally-distributed computers appear to work like a single machine. In the interests of minimal kernel size, configurability, and easy modification, many of the traditional functions of a monolithic operating system are placed outside the kernel in a

[2] To support a dispatcher, the concurrency mechanism must be designed in such a way that a particular thread can be made runnable explicitly. Languages that provide a static number of synchronization conditions (as, for example, in Modula-1 [29]) are not well suited to this purpose.

distributed operating system. In Charlotte, for example, user-level code is used to implement a process and memory manager (the *starter*), a command interpreter, a process connection facility, two kinds of file servers, a name server (the *switchboard*), and a terminal driver. The communication activity of these processes is at least as complicated, and often more complicated, than that of any single user application.

From the user's point of view, the presense of servers means that even the most self-contained of programs is likely to need to communicate occasionally with an independently-developed process about which it knows very little. Language support can make this communication much more convenient and safe, particularly if it matches the style of communication used *within* the program. In fact, the more one moves away from the centralized model of a traditional operating system and toward a distributed collection of servers, the harder it becomes to draw the line between one program and the next.

Consider a large-scale application that spans a geographically distributed collection of machines. An airline reservation system is an obvious example. It is possible to conceive of such an application as a single distributed program, but the concept wears thin when one considers the large number of institutions involved. It is one thing to talk about a program developed by a single organization (the airline, say) and running on machines owned and operated by that organization. It is quite another to talk about a program that has pieces under the control of a thousand different travel agencies, possibly written in different languages and running on different types of hardware. If we consider the subject of automatic teller machines and the electronic transfer of funds between competing, autonomous banks, the concept of a single distributed program breaks down altogether. As internets proliferate, the software required for network management and routing is also developing into a multi-program distributed application. Further examples can be found in the Defense Department's need for global information gathering and communication, and in similar applications in weather forecasting and ground control for spacecraft.

2.3. Special Language Needs

When compared to a multi-process program, the pieces of a multi-program application have unusually stringent needs for compile- and run-time flexibility, protection, and conversation management.

Flexibility

Operating system server processes must be able to communicate with clients that did not even exist when the server was designed. The obvious way to perform type checking for this communication is create for each server an interface description that can be used when constructing clients. Unfortunately, maintaining a consistent set of interface descriptions across distributed sites becomes a non-trivial database management problem as soon as there is a significant number of servers or sites. If the compiler insists that a client be compiled with exactly the same set of declarations that were used to compile the server (this amounts to insisting on *name equivalence* for types [12, pp. 134-136]), then the database problem becomes particularly severe: it requires unforgable version numbers that change when the data changes but not when it is copied.

Upward compatibility is also a problem. If a new comment is added to the interface description for the file server, we will certainly want to avoid recompiling every process that uses files. If a new routine is added without changing the behavior of the rest of the interface, we should also avoid recompilation. If changes are made to certain routines but not to others, we should recompile only those processes whose

behavior would otherwise be incorrect. It is possible to build a compiler that incorporates a formal notion of upward compatibility [26], but the task is not a simple one, even in the absence of multiple sites. Lynx addresses the problem by checking *structural* type equivalence, at run time, for each individual message: no central database of types is needed, and upward-compatible programs run without recompilation.

Servers must frequently change their interconnection topology at run time. To facilitate changes, Lynx makes the message channel between processes an explicit first-class object, the *link*, and provides variables that name a link. A link can be used to represent an abstract *resource* that is distinct from both the process(es) that implement it and the operations it provides. Examples of resources include a file, a bibliographic database, a print spooler, and a process creation facility.

Most other languages that allow connections to be reconfigured use variables that name processes or remote operations. The problem with naming processes is that a resource may be provided by a collection of processes: a distributed server may prefer that a user communicate with different constituent processes at different times, in order to balance workload or minimize communication costs. If users address their messages to processes the server cannot effect topology changes without informing the user, a violation of abstraction. The problem with naming remote operations is that a resource may provide different sets of operations to different clients, or at different points in time. Even with facilities for bundling related operations (such as the *resource capabilities* of SR [1]), the nature of each abstraction must be known to every client; servers cannot change the set of available operations to reflect changes in the state of the abstraction or to implement access control. If the resource is passed among clients, the desire for information hiding suggests that each client should be aware of only the operations it needs.

Since type checking is performed on messages in Lynx, connections between processes can be manipulated without knowing the types of messages that links may eventually carry. Name servers and other interconnection utilities can establish connections between processes whose message interfaces were created long after the servers were placed in operation. In a language with typed connections (particular one such as Ada, which addresses messages to operations) one cannot write a name server capable of registering clients that use newly-created message types.

Protection

Pieces of a multi-program application cannot afford to trust each other. Even if malice is not an issue (as a result, let us say, of external administrative measures), a healthy respect for the principles of software engineering dictates that each process be able to recover from arbitrary errors on the part of its communication partners. The easiest way to provide such protection from external errors is probably to incorporate message passing into a general-purpose exception-handling mechanism, with a built-in exception for each type of system-detectable error. It will always be possible, of course, for a process to send messages with incorrect data, but no language could prevent it from doing so. The types of errors that must be propagated up to the user are such language- and system-defined events as type clashes (with run-time type checking), use of an incorrect address, termination of a communication partner, or failure of a processor or channel.

On a more fundamental level, communication facilities for mutually suspicious processes must be designed in such a way that each process can exercise control over whom it talks to when. A server, for example, must be able to specify not only the clients to whom it wishes to send messages, but also the

clients from whom it is willing to receive. Even if it is willing to receive messages from anywhere, a server is still likely to need to differentiate between clients in order to provide them with differing levels of service or extend to them differing levels of trust. Few existing languages provide a receiver with this sort of expressive power.

Conversation Management

In a language with multiple threads of control, the concurrency between threads can be used for two quite different purposes. It can serve to express true parallelism, for the sake of enhanced performance, or it can serve as a program structuring tool to simplify the exposition of certain kinds of algorithms. In a server process the latter purpose may be particularly important; it captures the existence of independent, partially-completed conversations with multiple clients. Unless servers are expected to run on multiprocessor hardware, the goal of running threads in parallel may not be important at all. It is of course attractive to have a lightweight thread mechanism that addresses both goals at once, but it introduces the need for fine-grained synchronization on data that is shared between threads. In a monitor-based language with a stub generator (Cedar [25] for example), the programmer must keep track of two very different forms of synchronization: monitors shared by threads in the same address space and remote procedure calls between threads in different address spaces. Dissatisfaction with a similar approach in Washington's Eden project [5] was a principal motivation for the development of the Emerald language [4]. Emerald provides an object-oriented model that eliminates the distinction between local and remote invocations, but it still requires monitors to synchronize concurrent invocations within a single object. In SR [1], an alternative approach to unifying remote and local invocations allows a single style of synchronization to be used in all situations.

No matter how elegant the synchronization mechanism, however, its use is still a burden to the programmer. In an implementation without true parallelism, pseudo-concurrent semantics for threads create the appearance of race conditions that should not even exist. They force the use of explicit synchronization on even the most simple operations.[3] An alternative approach adopted in Lynx, is to abandon the possibility of true parallelism within processes in favor of simpler semantics. Threads in Lynx, like coroutines, run until they block. They are purely a program structuring tool.

In either case, whether threads are said to run in parallel or not, a process that is communicating with several peers at once can benefit tremendously from a careful integration of thread management with the facilities for passing messages. If each conversation with a client is to be represented by a separate thread of control, a server may wish to arrange for new threads to be created automatically in response to certain kinds of incoming messages. Such *implicit receipt* of messages by a newly created thread is characteristic of concurrent languages with stub generators, and is also found in Argus, Emerald, and SR. In Lynx it is extended to permit the creation of threads in a nested lexical context, so that related threads can share state. In a language with coroutine-like threads, remote procedure calls and other blocking communication statements can produce an automatic coroutine transfer to another, runnable thread.

[3] To increment a shared variable, for example, a pseudo-concurrent thread need not worry about atomicity. It can assume that a context switch will not occur between its read and write. If we pretend that threads are truly parallel, then the program will not be "correct" unless we write code to specify that the physically atomic increment operation should also be semantically atomic.

3. Language Design

3.1. Links

Processes in Lynx are assumed to be independent and autonomous. Each process is separately compiled. At run time, processes communicate only by sending messages to each other over two-directional communication channels called *links*. Each process begins with an initial set of arguments, presumably containing at least one link to connect it to the rest of the world. Each link has a single process at each end. As an example of a simple application, consider a producer process that creates data of some type and sends that data to a consumer. Each process begins with a link to the other. The producer looks like this:

```
process producer (consumer : link);

type data = whatever;
entry transfer (info : data); remote;

function produce : data;
begin
    -- whatever
end produce;

begin  -- producer
    loop
        connect transfer (produce |) on consumer;
    end;
end producer.
```

The word "entry" introduces a template for a remote operation. The general syntax is

```
entry opname ( request_parameters ) : reply_parameter_types ;
```

In this case, the transfer entry has no reply parameters.

An entry header can be followed by a body of code, or by the word "remote." In our case, we have used the latter option because the code for transfer is in another process. Like the word "forward" in Pascal, "remote" can also indicate that the code will appear later in the current process, either as a repeated entry declaration or as the body of an *accept* statement, as in the consumer below.

The connect statement is used to request a remote operation. The vertical bar in the argument list separates request and reply parameters.

```
connect opname ( expr_list | var_list ) on linkname ;
```

The current thread of control in the sending process is blocked until a reply message is received, even if the list of reply parameters is empty. Our producer has only one thread of control (more complicated examples appear below), so in this case the process as a whole is blocked.

The consumer looks like this:

```
process consumer (producer : link);

type data = whatever;
entry transfer (info : data); remote;
```

```
procedure consume (info : data);
begin
    -- whatever
end consume;

var buffer : data;

begin  -- consumer
    loop
        accept transfer (buffer) on producer; reply;
        consume (buffer);
    end;
end consumer.
```

The accept statement is used to provide an operation requested by the process at the other end of a link. In our example, the producer uses a connect statement to request a transfer operation over its link to the consumer, and the consumer uses an accept statement to provide this operation.

```
accept opname ( var_list ) on linkname ;
    ...
reply ( expr_list ) ;
```

The reply clause at the end of the accept statement returns its parameters to the process at the other end of linkname and unblocks the thread of control that requested the operation opname. The parameter types for opname must be defined by an entry declaration. Arbitrary statements can appear inside an accept statement, including nested accepts. In our example, the consumer has nothing it needs to do before replying.

A link can be thought of as a *resource*. In our example neither the consumer nor the producer can name the other directly. Each refers only to the link that connects them. The consumer, having received all the data it wants, might pass its end of the link on to another process. Future transfer operations would be provided by the new consumer. The producer would never know that anything had happened.

A variable of type link really identifies a link *end*. Link ends are created in pairs, by a built-in routine called newlink. Our producer/consumer pair could be created with the following sequence of statements:

```
var L : link;
begin
    startprocess ("consumer", newlink (L));
    startprocess ("producer", L);
    ...
```

To make it easy to write sequences of code such as this one, newlink returns one of the link ends as its function value (here passed on immediately to the consumer) and the other through a reference parameter (here saved temporarily in L so that it can be passed to the producer in the second call to startprocess).

Since messages are addressed to links, not processes, it is not even necessary to connect the producer and consumer directly. An extra process could be interposed for the purpose of filtering or buffering the data. Neither the producer nor the consumer would know of the intermediary's existence.

```
startprocess ("consumer", newlink (L));
startprocess ("buffer", L, newlink (M));
startprocess ("producer", M);
```

Code for a buffer process appears in section 3.4.

To move a link end, a process need only enclose it in a message (via connect, accept, reply, or startprocess). Once the message is received, the sending process can no longer use the transferred link, but the receiving process can. The compiler provides the run-time system with enough information about types that this transfer is guaranteed to work for messages containing arbitrary data structures (including variant records) that might have links inside.

A link between a server and a client can be passed on to a new client when the first one doesn't need it any more. It can also be passed on to a new *server* (functionally equivalent to the old one, presumably) in order to balance work load or otherwise improve performance. In a large distributed environment, many servers are likely to be implemented by *collections* of processes. These processes may move their end of a client link frequently, in order to connect the client to the member of their group best able to serve its requests at a particular point in time.

One common use of link movement is to implement a *name server* process that maintains a database of server names and links. Clients in need of a particular service can ask the name server for a link on which to request that service (see box, p. 11).

3.2. Implicit Receipt

In our producer/consumer example, each process contains a single thread of control. In the consumer, this thread accepts its transfer requests explicitly. It is also possible to accept requests *implicitly*, and the choice between the two approaches depends largely on whether we view the consumer as an active or a passive entity.

If we think of the producer and consumer as active peers, then it makes sense for the consumer to contain a thread that "deliberately" waits for data from the producer. If we choose, however, to think of the consumer as a server (a spooler for a printer, perhaps), then we will most likely want to write a more passive version of the code — one that is driven from outside by the availability of data. Since a demand-driven spooler is likely to have multiple clients, it also makes sense to give each incoming request to a separate thread of control, and to create those threads automatically. Our consumer can be re-written to use implicit receipt as follows:

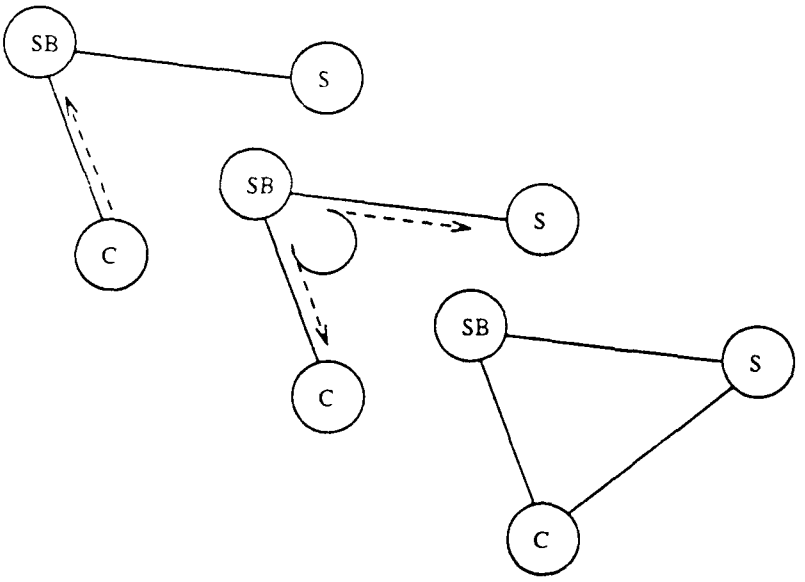
```
process consumer (producer : link);

type data = whatever;

procedure consume (info : data);
begin
    -- whatever
end consume;
```

Link Movement

There are many reasons to change the connections between processes at run time. Certainly a newly-created process must be connected to existing processes that can provide it with input and output. In a single, large application, it is also common for computation to move through a series of distinct phases, each of which requires a different set of processes and connections. In a robust, geographically distributed system, a process that is unable to obtain a service from its usual source (because of hardware failures, for example, or overloaded communication lines) may wish to connect to an alternative server. In the example below we see a process (called the *switchboard*) whose job it is to keep a registry of servers who are willing to accept new clients. The command interpreter, or shell, is likely to provide each newly-created process with a link to the switchboard, from which it can obtain links to whatever other servers it may need.



To find, for example, a line-printer spooler, client C would send a message to the switchboard:

```
connect find_server ("lp_spooler" | spooler_link) on switchboard_link;
```

Upon receiving this request, the switchboard would scan its registry for a server that has advertised the name "lp_spooler." Assuming such a server exists, the switchboard would create a new link, pass one end on to the server in a newclient message, and return the other end to the client:

```
entry find_server (server_name : string) : link;
var server, rtn : link;
begin
      server := lookup (server_name);
      if server = nolink then reply (nolink);
      else connect newclient (newlink (rtn) |) on server; reply (rtn);
      end;
end find_server;
```

Newlink returns one link end as its function return value (which here becomes a parameter to the newclient operation) and the other through a reference parameter. Each server that wishes to accept new clients must provide the switchboard with a link over which it is willing to accept newclient requests.

```
entry transfer (info : data);
begin
    reply;
    consume (info);
end transfer;

begin  -- consumer
    bind producer to transfer;
end consumer.
```

Here we have provided a begin ... end block for the transfer entry procedure, instead of declaring it remote. Each connect to transfer will create a new thread of control in this version of the consumer. As with an accept block, the reply statement of the entry causes the run-time support package to unblock the thread of control (in the producer) that requested the current operation. The replying thread continues to exist until it runs off the end of its entry. The producer shown above can be used with either version of the consumer, without modification.

The bind statement serves to create an association between links and entry procedures:

```
bind link_list to entry_list ;
```

Only those operations provided by accept statements and bindings to entries can be requested by the process at the far end of a link. Connect statements that request a non-existent operation will cause an Ada-like exception in the requesting thread of control.

Bindings can be broken as well as made:

```
unbind link_list from entry_list ;
```

The ability to manipulate bindings at run time is a powerful mechanism for access control. Each process has complete control over which of its communication partners can invoke which operations at which points in time. Reference [20] contains a Lynx solution to the classic reader/writers problem. This solution permits a client to obtain read and/or write access to a resource and perform an arbitrary sequence of operations before relinquishing that access. The sequence of operations need not be known at the time that access is obtained; a client can, for example, obtain read access, read an index, and read a location calculated from that index in one protected session. A similar solution in Ada [28] requires a complicated system of unforgable *keys*, implemented in user code.

It is the ability of a server to refer to links by name that permits it to implement access control. A server can, if desired, consider clients as a group by gathering their links together in a set and by binding them to the same entries. It is never forced, however, to accept a request from an arbitrary source that happens to know its address. Of course, a server has no way of knowing which *process* is attached to the far end of a link, and it has no way of knowing when that far end moves, but this is in keeping with the concept of process autonomy. A link to a client represents an abstraction (a connection over which to provide a service) every bit as much as a link to a server represents a connection over which a service is provided. In fact, it is entirely possible for two processes to act as servers for each other, with a single link between them. A file server, for example, might use a link to a sort utility in order to maintain indices. The sort utility for its part might use the file server as a place to store large data sets.

Symmetric, two-directional links strike a compromise between absolute protection on the one hand and simplicity and flexibility on the other. They provide a process with complete run-time control over its connections to the rest of the world, but limit its knowledge about the world to what it hears in messages. A process can confound its peers by restricting the types of requests it is willing to accept, but the consequences are far from catastrophic. Exceptions are the most serious result, and exceptions can be caught. Even an uncaught exception kills only the thread that ignores it.[4]

3.3. Type Checking

To a large extent, links are an exercise in delayed decision making. Since the links in communication statements are variables, requests are not bound to communication paths until the moment they are sent. Since the far end of a link can be moved, requests are not bound to receiving processes until the moment they are received. Since the set of valid operations depends on outstanding bindings and accepts, requests are not bound to receiving threads of control until after they have been examined by the receiving process. Only after a thread has been chosen can a request be bound to the types it must contain. Checks must be performed on a message-by-message basis.

Run-time type checking provides three distinct advantages:

(1) A process can hold a large number of links without knowing what types of messages they may eventually carry. A name server, for example, can keep a link to each registered process, even though many such processes will have been created long after the name server was compiled and placed in operation.

(2) A process can use the same link for different types of messages at different times, or even at the same time. It need not declare a conservative superset of those types at compile time, nor ever worry about receiving a message of a currently-inappropriate type at run time.

(3) With an appropriate choice of semantics for type equivalence, the compiler can be designed to work without a global database of types. The language implementor is relieved of the burden of distributed database management and of the problem of upward compatible changes.

Type checking in Lynx is based on *structural* equivalence. Two types are considered the same if they contain the same internal structure — the same set of primitive types composed with same higher-level type constructors. The compiler can provide the run-time system with a "canonical" representation of each type, so that type checking becomes a simple comparison for equality of canonical forms.

Since canonical forms can be of arbitrary length, run-time comparisons are potentially costly. To minimize this cost, the Lynx compiler uses a hash function to compress its type descriptions into 32-bit codes [22]. Hashing reduces the cost of type checking to less than 10 microseconds per remote operation. It introduces the possibility of undetected type clashes, but at a probability of less than one chance in a billion with a good hash function.

A second potential problem with run-time checking is that programming errors that would have been caught at compile time in other languages may not be noticed until run time in Lynx. This cost, too, is

[4] Admittedly, a malicious process can serve requests and provide erroneous results. No language can prevent it from doing so.

small, and easily justified by the type system's simplicity and flexibility. As a practical matter, we tend to rely on shared declaration files to ensure that run-time clashes are rare. We catch most type errors at compile time and the rest at run time, much more easily than we could catch all of them at compile time.

A final cost of the Lynx approach to types is the somewhat liberal checking implied by structural equivalence. Variables with the same arrangement of components will be accepted as compatible even if the abstract meanings of those components are unrelated. Lynx shares this form of checking with many other languages, including Algol-68, Smalltalk, Emerald, and many dialects of Pascal. We are happy with structural equivalence. No type system, no matter how exacting, can *ensure* that messages are meaningful. Type checking can be expected to reduce the likelihood of data misinterpretation, not to eliminate it.

3.4. Using Multiple Threads

Though the implicit-receipt version of our consumer process will contain a thread for every invocation of the transfer operation, it is likely that only one such thread will exist at a time. For a slightly more complicated example, consider the buffer process mentioned above. Interposed between a producer and consumer, the buffer serves to smooth out fluctuations in their relative rates of speed.

```
process buffer (consumer, producer : link);
const size = whatever;
type data = whatever;
var
    buf : array [1..size] of data;
    firstfree, lastfree : [1..size];

entry transfer (info : data);
begin
    await firstfree <> lastfree; -- not full
    buf[firstfree] := info;
    firstfree := firstfree % size + 1;
    reply;
end transfer;

var info : data;
begin
    firstfree := 1;
    lastfree := size;
    bind producer to transfer;
    loop
        await lastfree % size + 1 <> firstfree; -- not empty
        lastfree := lastfree % size + 1;
        info := buf[lastfree];
        connect transfer (info |) on consumer;
    end;
end buffer.
```

Every Lynx process begins with a single thread of control, executing the process's main begin ... end block. New threads are created in response to incoming requests on links bound to entries, and may also be created explicitly by "calling" an entry locally.

The threads of control within a single process do not execute in parallel; each process continues to execute a single thread until it *blocks*. The process then takes up some other thread where it last left off. If no thread is runnable, then the process waits for completed communication to change that situation.

Threads may block (1) for communication (connect, accept, reply), (2) for completion of nested threads (when leaving a shared scope), (3) for a reply from a locally-created thread, and (4) for an explicitly await-ed condition. In the bounded buffer example, the await statement blocks the current thread until the buffer is non-empty or non-full, as appropriate. There is no need to worry about simultaneous access to buf, firstfree, or lastfree, because the coroutine-style semantics guarantee that only one thread can execute at a time.

Of course, the mutual exclusion of threads in Lynx prevents race conditions only between context switches. In effect, Lynx code consists of a series of critical sections, separated by blocking statements. Since context switches can occur inside subroutines, it may not be immediately obvious where those blocking statements are, but the compiler can help by identifying them in listings. Experience to date has not uncovered a serious need for inter-thread synchronization across blocking statements. For those cases that do arise, a simple Boolean variable in an await statement performs the work of a semaphore.

The syntax of Lynx allows entries to be declared at any level of lexical nesting. Non-global data may therefore be shared by more than one thread of control. The run-time data structures required to implement this sharing are discussed in the box on page 16. In the file server example in the following section, there will be one thread of control for every open file. Additional, nested threads will be used to implement file-specific operations.

A link end may be bound to more than one entry. The bindings need not be created at the same time. A bound end can even be used in subsequent accept statements. These provisions make it possible for separate threads to carry on independent conversations on the same link at more or less the same time. The startprocess statement, for example, might be implemented by sending a request to a process manager written in Lynx. Each such request might create a new thread of control within that manager. Separate threads could share the same link between the process manager and file server. Their requests to open and read executable files would interleave transparently.

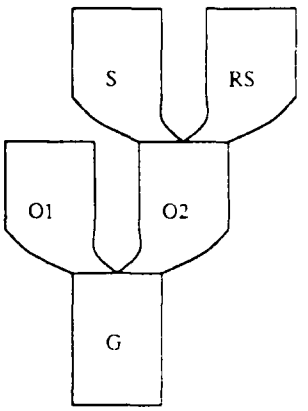
When all of a process's threads are blocked, run-time support routines attempt to receive a message on any of the links for which there are outstanding accepts or bindings, or on which replies are expected for outstanding connects. Incoming replies can only have been sent in response to an outgoing request. Each such reply can therefore be delivered to an appropriate thread of control. Incoming replies, by contrast, can be unexpected or unwanted. The operation name of a request is compared against those of the outstanding accepts and bindings for its link. If a match is found, then an appropriate thread can be made ready and execution can continue. If there are no accepts or bindings, then consideration of the message is postponed. If accepts or bindings exist, but none of them match the request, then the message is discarded and an exception is raised in the thread that executed the connect statement at the other end of the link.

3.5. A File Server Example

A more realistic example of the use of threads and links can be seen in figure 1. This figure contains a simplified version of the code for a file server process under Charlotte. The original Charlotte file server was written in a dialect of Modula, using sequential features only, and relying on library calls for

Cactus Stacks

A **cactus stack** is actually a *collection* of stacks, prefixes of which may be shared. The most common use of cactus stacks is to manage space for nested coroutines. Although the exact origin of the concept is obscure, it dates to at least the middle 1960's, when it appeared in the architectures of the Burroughs B6500 and B7500 computers.



In the file server example of figure 1, consider what must happen when two different files are open and one of the threads that manage those files contains nested threads itself. Both of the threads executing the open entry procedure will have access to global variables. They will, of course, have access to their own local variables as well. Now if one of those threads (call it 'O2') is managing a file that is open for both reading and writing (with random access), it will be possible for incoming stream and readseek requests to create concurrent threads inside the context of O2 (concurrent in the sense that they exist and occupy space simultaneously). These threads will need to share access to O2's local variables.

In the illustration, each segment of the cactus stack represents a subroutine or entry activation. Each branching point corresponds to the creation of a nested thread of control. There is therefore one thread per leaf of the structure, and one thread for every internal segment (other than the root) that lies at the top of a trunk. From the point of view of any one thread, the path back down to the root looks like a normal stack.

Lynx ensures that no thread can leave a scope in which nested threads or bindings are still active. This rule ensures that segments never "disappear" out of the middle of the structure. In our file server example, a thread executing the open entry for a write-only file will finish its work almost immediately. When it reaches the end of its scope, however, it will be suspended automatically until there is no further possibility of creating threads in the writeseek, stream, and readseek entries. This will occur only when the client closes its file by destroying the link that represents it.

Each trunk of a cactus stack can be placed in a single, contiguous array, allocated when its thread is created. Alternatively, each individual segment (frame) can be allocated dynamically. The Lynx compiler adopts the latter approach in order to economize on space, in the expectation that programs may contain very large numbers of threads. A caching strategy for frames makes allocation relatively inexpensive, but further optimization is still desirable. As it parses its input files, the Lynx compiler keeps track of which subroutines contain statements that may cause a thread switch. Space for such routines must be allocated in the cactus stack, since returns need not occur in LIFO order. For routines that *cannot* cause a thread switch, however, space may be allocated on an ordinary stack. In practice most subroutines (and particularly those that are frequently called) can be seen to be sequential. The coroutine-like semantics of threads in Lynx allow these routines to be implemented at precisely the same cost as in conventional sequential languages.

---

communication. It consisted of just under 1000 lines of code, and was written and rewritten several times over the course of a two-year period. It was a constant source of trouble. The Lynx fileserver is just over 300 lines, and was written in only two weeks. It would have required even less time if it had not been undertaken concurrently with debugging of the language implementation.

```
1  process fileserver (switchboard : link);
2  type string = whatever; bytes = whatever;

3  entry open (filename : string; readflag, writeflag, seekflag : Boolean) : link;
4  var filelnk : link; readptr, writeptr : integer;
5  exception seeking;

6      procedure put (data : bytes; filename : string; writeptr : integer);
7          external;
8      function get (filename : string; readptr : integer) : bytes; external;
9      function available (filename : string) : Boolean; external;

10     entry writeseek (newptr : integer);
11     begin
12         writeptr := newptr; reply;
13     end writeseek;

14     entry stream (data : bytes);
15     begin
16         put (data, filename, writeptr); writeptr +:= 1; reply;
17     end stream;

18     entry readseek (newptr : integer);
19     begin
20         readptr := newptr; announce seeking; reply;
21     end readseek;

22 begin  -- open
23     if available (filename) then
24         reply (newlink (filelnk));  -- release client
25         readptr := 0; writeptr := 0;

26         if writeflag then
27             if seekflag then bind filelnk to writeseek; end;
28             bind filelnk to stream;
29         end;

30         if readflag then
31             if seekflag then bind filelnk to readseek; end;
32             loop
33                 begin
34                     connect stream (get (filename, readptr) | ) on filelnk;
35                     readptr +:= 1;
36                 when seeking do
37                     -- nothing; try again at new location
38                 when REMOTE_DESTROYED do
39                     exit;  -- leave loop
40                 end;
```

```
41              end;  -- loop
42          end;  -- if readflag
43      else  -- not available
44          reply (nolink);  -- release client
45      end;
46      -- control will not leave 'open' until nested entries have died
47  end open;

48  entry newclient (client : link);
49  begin
50      bind client to newclient, open; reply;
51  end newclient;

52  begin  -- main
53      bind switchboard to newclient;
54  end fileserver.
```

Figure 1: Stream-based file server in Lynx.

The newclient convention has been used in this example. We have written the server to take a single initial argument: a link to the switchboard name server. Additional clients are introduced by invocations of newclient over links from the switchboard or from clients. When a newclient request is received (line 48), the file server binds that link to an entry procedure for each of the services it provides. One of those entries, for opening files, is shown in this example (lines 3-47).

Open files are represented by links. Within the server, each file link is managed by a separate thread of control. New threads are created in response to open requests. After verifying that its physical file exists (line 23), each thread creates a new link (line 24) and returns one end to its client. It then binds the other end to appropriate sub-entries. Among these sub-entries, context is maintained automatically from one request to the next. We have adopted the convention that data transfers are initiated by the producer (with connect) and accept-ed by the consumer. As we have seen, this asymmetry allows the transparent insertion of an intermediate filter or buffer. When a file is opened for writing the server plays the role of consumer. When a file is opened for reading the server plays the role of producer.

In additional to a conventional mechanism for raising exceptions in a single thread of control, Lynx also permits one thread to cause an exception in another. In the file server example, this facility is used to handle seek requests in a file that is open for reading. Under the normal stream protocol, the file server will always attempt to transfer a block (with "connect stream ...") as soon as the previous block has been received. In order to read blocks out of order, the client invokes a readseek operation. The thread that provides this operation uses an announce statement (line 20) to interrupt the thread (line 36) that is trying to send the wrong block. That thread then retries its connect, using the updated file pointer. Since incoming requests (and readseek requests in particular) are received only when all threads are blocked, the thread that provides the readseek operation can be sure that the thread that is streaming data must be stopped at its connect statement. No race conditions can occur.

To close an open file, a client need only destroy the link that represents the file.[5] A thread that tries to use a destroyed link feels a REMOTE_DESTROYED exception (caught at line 38 in the file server). Bindings for a destroyed link are broken automatically. These mechanisms suffice in this example to clean up the context for a file.

3.6. A Distributed Game-Playing Program

Using an implementation of Lynx on the BBN Butterfly multiprocessor, we have created a distributed program that plays the game of checkers (draughts). Since our principal goal was to evaluate Lynx and not to investigate the design of parallel algorithms, we adopted an existing parallelization of alpha-beta search, designed by Jack Fishburn at Wisconsin [11].

The basic idea behind the algorithm can be seen in figure 2. There are three different kinds of processes. One process (the "master") manages the user interface (in our case, this is a graphic display under the X window system). A second process (the "midling") manages the parallel evaluation of possible moves. A third kind of process (the "slave") performs work on behalf of the midling. There is only one master and one midling. Performance is maximized when there is one slave for every available processor.

Within the midling, one thread of control is dedicated to exploring the first few levels of the game tree. It constructs a data structure describing all of its possible moves, all of the possible subsequent moves by its opponent, all of its possible moves after that, and so forth. At a given depth in the tree (typically four or five levels), it enters board positions into a queue of work to be performed by slaves.

Each slave is represented by a separate thread in the midling. That thread repeatedly removes an entry from the work queue, sends it to a slave, and waits for the result. When that result comes back it updates the game tree, performs any necessary pruning (to throw away moves that are now known to be sub-optimal), and obtains a new entry from the queue. In order to avoid storing all of the top few levels of the (very large) game tree at once, the thread that creates the data structure blocks when the work queue is full. Games tree nodes are thus created on demand. Likewise, the threads that dispatch work to slaves will block when the work queue is empty. Despite the fact that the checkers player is a single, coordinated program, the midling bears a strong resemblance to a server. It would have been significantly more difficult to write the midling with a single thread of control.

One consequence of the communication semantics of Lynx is that a process does not notice incoming messages until all of its threads are blocked. There is no way to receive a message asynchronously or to allow a high-priority message to interrupt the execution of lower-priority "background" computation. In the checkers program, performance is likely to improve if a slave can be interrupted when the midling discovers that its subtree has been pruned, or perhaps when it discovers new information that will help the slave do more pruning internally. For cases such as this, Lynx provides a low-cost polling function that can be used to determine if messages are pending. Slaves execute the statement

---

[5] Destroy is a built-in procedure that takes a single parameter of type link. Variables accessing either end of a destroyed link become dangling references.

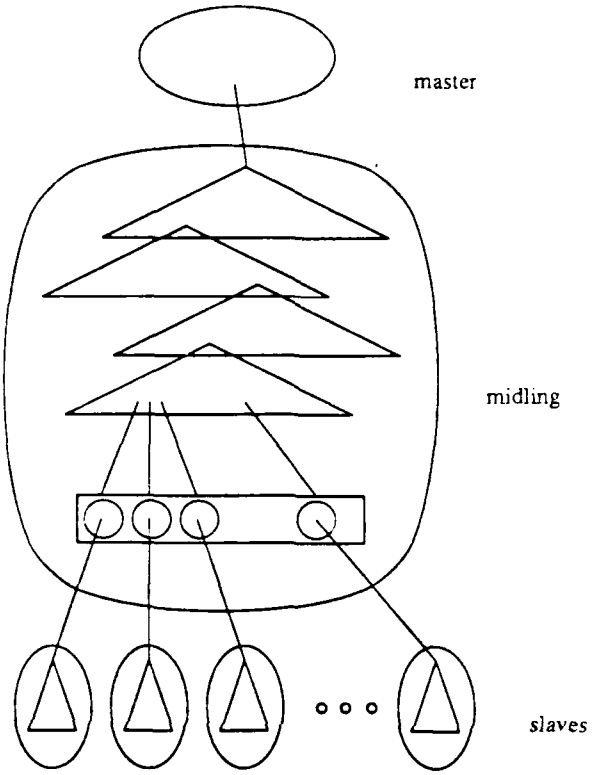


Figure 2: Checkers program structure.

```
await idle;
```

at the top of an outer loop. Update messages from the midling are therefore received within a reasonable amount of time.

We have yet to perform an extensive performance study on the checkers-playing program. Informal experiments with various problem parameters (number of tree levels in the midling, size of subtrees evaluated by slaves, frequency of update messages, etc.) have produced approximately 20-fold speedups with 100 working slaves. The primary limiting factor appears to be that many of the subtrees that are evaluated in parallel would have been pruned off and never explored by the standard sequential algorithm.

4. Implementation Experience

An implementation of Lynx for Wisconsin's Charlotte operating system was completed in 1984. It was ported to a simpler version of Charlotte in 1987. Charlotte runs on a collection of VAXen connected by a token ring [9]. An implementation for the BBN Butterfly multiprocessor was completed in 1986. Other, paper designs exist for UNIX (using TCP/IP) and for an experimental system known as SODA [14]. In addition to providing a testbed for evaluating Lynx, our implementation experience has led to

unexpected insights into the relationship between a language run-time package and the underlying operating system [11, 20], and also into the factors that contribute to message passing overhead [22].

4.1. The Language/Kernel Interface

A distributed operating system provides a process abstraction and primitives for communication between processes. A distributed programming language can regularize the use of the primitives, making them both safer and more convenient. The level of abstraction of the primitives, and therefore the division of labor between the operating system and the language support routines, has serious ramifications for efficiency and flexibility. Lynx is one of the few distributed languages that has been implemented on top of more than one operating system.

Simply put, the implementation experience with Lynx is that the more primitive the operating system (within reason) the easier it is to build a language above it. When we set out to implement Lynx we did not expect to discover this result. Symmetric, two-directional links are directly supported by Charlotte. The original motivation for Lynx was to build a language around them. Yet despite the fact that Charlotte kernel calls provided links as a fundamental abstraction, the implementation of Lynx was extremely complicated and time-consuming. Many of the functions provided by the kernel were almost, but not quite, what the run-time package needed. For example, Charlotte's *receive* function provided no way to say that only reply messages were wanted (and not requests). A complicated protocol was required in the run-time package in order to reject and return unwanted requests. Similarly, the Charlotte *send* function allows links to be enclosed in messages, but only one at a time. Additional run-time protocol was required to packetize multi-link messages.

By comparison, implementation of Lynx on top of Chrysalis was surprisingly easy, despite the fact that Chryalsis has no notion of a link or even of a message. What Chrysalis *does* provide are low-level facilities for creating shared memory blocks and for atomically manipulating flags and queues. The box on page 22 explains how links can be built from these primitives. The fact that Chrysalis supports shared memory is a significant but not deciding factor in it suitability for Lynx. Our paper implementation for the message-based primitives of SODA is equally simple. Our TCP/IP design lies somewhere in the middle.

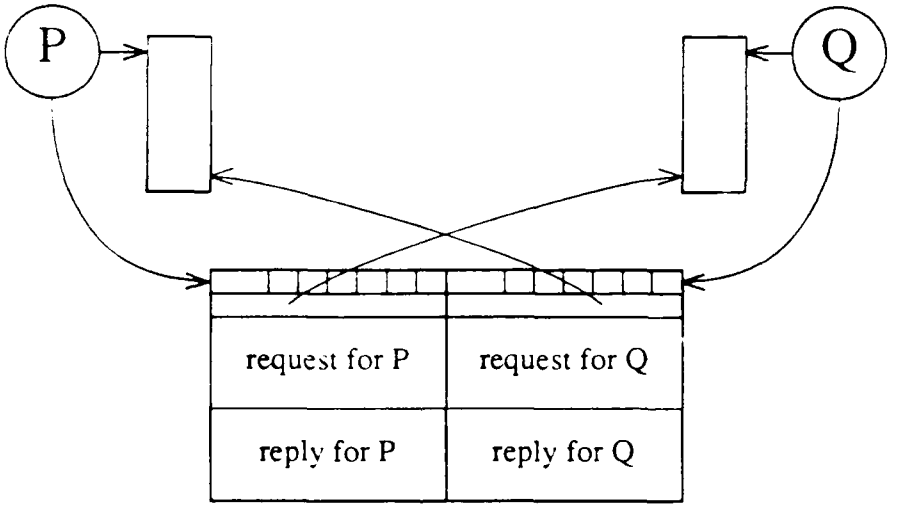
Like most distributed operating systems, Charlotte was designed with the expectation that programmers would invoke its primitives directly. This expectation appears to have been naive, but by no means unique. The proliferation of remote procedure call stub generators suggests that users of a wide range of message-passing operating systems have found their primitives too primitive to use. Unfortunately, the creation of interfaces that are *almost* usable for day-to-day programming has meant that substantial amounts of functionality and, consequently, flexibility, have been hidden from the user. Remote procedure calls may work, but alternative approaches to naming, buffering, synchronization, error recovery, or flow control are generally not available.

Our experience with Lynx suggests that an operating system kernel should either be designed to support a single high-level language (as, for example, in the dedicated implementations of Argus [17], Linda [7], and SR [1]), or else should provide only the lowest common denominator for things that will be built upon it. A middle-level interface is likely to be both awkward and slow: awkward because it has sacrificed the flexibility of the more primitive system; slow because it has sacrificed its simplicity. We recommend low-level kernels because they can maintain flexibility without introducing costs.

Links Under Chrysalis

The Butterfly implementation of Lynx consists of a cross compiler that runs on a host machine and a run-time support package that implements links in terms of Chrysalis primitives. For compatibility reasons, and to simplify the implementation, the compiler generates C for "intermediate code". Errors in the Lynx source inhibit code generation, so the output, if any, will pass through the C compiler without complaint. Programmers are in general unaware of the C back end.

On the Butterfly, every Lynx process allocates a Chrysalis atomic queue when it first begins execution. This queue is used to receive notifications of messages sent and received on any of the process's links. A link is represented by a block of shared memory, mapped into the address spaces of the two connected processes. The shared memory object contains buffer space for a single request and a single reply in each direction. Since dynamic allocation and re-mapping of message buffers would be prohibitively expensive, messages are limited to a fixed maximum length, currently 2000 bytes. Each process keeps an internal list of the threads that are waiting for buffers to be emptied or filled.



In addition to message buffers, each link object also contains a set of flag bits and the names of the atomic queues for the processes at each end of the link. When a process gathers a message into a buffer or scatters a message out of a buffer into local variables, it sets a flag in the link object (atomically) and then enqueues a notice of its activity on the atomic queue for the process at the other end of the link. When all of the process's threads are blocked, it attempts to dequeue a notice from its own atomic queue, waiting if the queue is empty.

The flag bits permit the implementation of link movement. Whenever a process dequeues a notice from its atomic queue it checks to see that it owns the mentioned link end and that the appropriate flag is set in the corresponding object. If either check fails, the notice is discarded. Every change to a flag is eventually reflected by a notice on the appropriate atomic queue, but not every queue notice reflects a change to a flag. A link is moved by passing the (address-space-independent) name of its memory object in a message. When the message is received, the sending process removes the memory object from its address space. The receiving process maps the object *into* its address space, changes the information in the object to name its own atomic queue, and then inspects the flags. It enqueues notices on its own queue for any of the flags that are set.

4.2. The Cost of Message-Passing

In our Butterfly implementation of Lynx, the simplest remote operations complete in less than two milliseconds. To place this figure in perspective, a call to an empty procedure takes 10 microseconds on an individual Butterfly node. An atomic test-and-set operation on remote memory takes 35 μs. An atomic enqueue or dequeue operation takes 80 μs. In the following table, nullop is a trivial remote operation with no parameters. Bigop is the same as nullop, but includes 1000 bytes of parameters in each direction. Explicit receipt uses an accept statement; implicit receipt uses a binding to an entry.

	Explicit receipt: process nodes		Implicit receipt: process nodes	
	different	same	different	same
nullop	1.80 ms	2.58 ms	2.04 ms	2.76 ms
bigop	3.45 ms	4.21 ms	3.72 ms	4.42 ms

Inter-node operations finish more quickly than intra-node operations because the two processors can overlap their computations. Implicit receipt costs more than explicit receipt because of the need to create and destroy a thread. We have reason to believe that these times could be reduced by additional tuning, but it seems unlikely that the lowest figure would drop below a millisecond and a half. A remote invocation is thus two orders of magnitude more expensive than a local procedure call, a result that is consistent with most other well-tuned message-passing systems.

21% – actual communication
Clearing and setting flag bits, posting notices on queues, calculating locations of message buffers.

22% – thread management
Thread queue management, queue searching (dispatcher), context switches, cactus stack frame allocation, buffer acquisition.

11% – bookkeeping
Keeping track of which threads want which sorts of services and which threads are willing to provide them.

18% – checking and exception handling
Verifying link validity, verifying success of kernel calls, type checking, establishment of Lynx exception handlers, initialization of stack frame exception information.

6% – protocol option testing
Checking for link movement, asynchronous notifications, premature requests, optional acknowledgments.

22% – miscellaneous overhead
Timing loop overhead, dispatcher loop and case statement overhead, procedure-call linkage, caching of constants in registers.

Figure 3: Contributors to Message-Passing Overhead (in percent of total work performed)

Like many researchers, we found the cost of message passing to be both frustrating and puzzling. Not only did we wish that things worked faster, we also didn't *understand* why they worked at the speed they did. In order to obtain a better explanation of "where the time goes," we profiled benchmark programs at the instruction level and assigned each individual instruction to one of 23 different functional categories. The results of this profiling are summarized in figure 3. A timeline for a 2.0 ms remote operation appears in figure 4. The timeline indicates the amount of time devoted to each of the phases of a remote invocation, but provides relatively little insight into the expense of individual language features involved in message passing.

Procedure call overhead and flag bit manipulation are the only single items in the profiling table that account for more than 10% of the total communication overhead. Small savings could undoubtedly be realized here and there, but there does not seem to be any way to achieve significant performance gains without eliminating language features. Work by other researchers tends to confirm the hypothesis that data transmission times do not dominate the cost of practical message-passing systems [8, 14, 17, 23]. High-level semantic functions such as addressing, dispatching, bookkeeping, testing, and error handling are at least as significant and often more so. One millisecond appears to be a nearly universal lower bound on round-trip communication times with mid-1980s microprocessor-based architectures, suggesting that it may be impossible to provide attractive message-passing semantics in significantly fewer than 1000 instructions.

5. Conclusion

Numerous programs have been written in Lynx over the course of the past four years, both as research projects and as coursework [6, 9]. In comparison to sequential programs that perform communication through library routines, Lynx programs are consistently shorter, easier to debug, easier to write, and easier to read. Much of the explanation is simply the difference between a language and the lack thereof; Ada, Argus, Linda, NIL, and SR can make similar claims. With a few exceptions, all these languages provide attractive syntax, secure type checking, error handling with exceptions, and automatic management of context for multiple conversations.

Beyond these facilities, however, Lynx provides an unusual degree of run-time flexibility.

- Symmetric communication links provide abstraction and transparent reconfiguration without the restrictions of compile-time type checking.
- The ability to distinguish between clients provides access control and protection.
- Mutually-exclusive threads provide context for multiple conversations without the complexity of synchronizing access to memory. The integration of threads with communication combines the conceptual clarity of remote procedure calls with the performance of non-blocking messages.

Each of these advantages can be of use in single-program applications. Even more important, they extend the advantages of language support to autonomous but interacting programs.

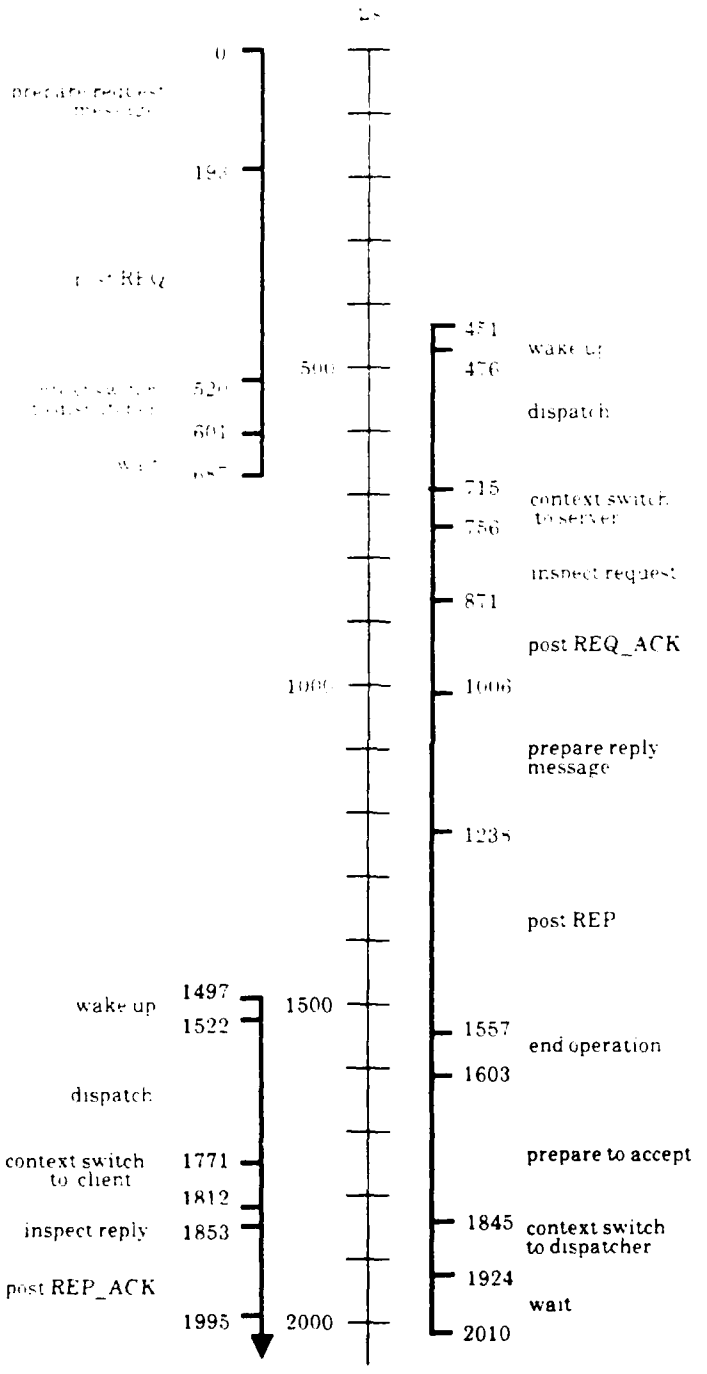


Figure 4: Timeline for a simple remote invocation.

Acknowledgments

Raphael Finkel supervised the Ph. D. thesis in which Lynx was originally defined. Marvin Solomon provided additional guidance. Ken Yap helped to port Lynx to the Butterfly. Alan Cox assisted with performance studies. Tom Virgilio ported Lynx to the simplified version of Charlotte. Bill Bolosky helped design the TCP/IP implementation. Numerous students, both at Wisconsin and at Rochester, provided feedback on the language and compiler.

At the University of Wisconsin, this work was supported in part by NSF CER grant number MCS-8105904, DARPA contract number N0014-82-C-2087, and a Bell Telephone Laboratories Doctoral Scholarship. At the University of Rochester, this work was supported in part by NSF CER grant number DCR-8320136, DARPA/ETL contract number DACA76-85-C-0001, and an IBM Faculty Development Award.

References

[1] G. R. Andrews, R. A. Olsson, M. Coffin, I. J. P. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, "An Overview of the SR Language and Implementation," *ACM TOPLAS 10*:1 (January 1988), pp. 51-86.

[2] Y. Artsy, H.-Y. Chang, and R. Finkel, "Interprocess Communication in Charlotte," *IEEE Software 4*:1 (January 1987), pp. 22-28.

[3] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM TOCS 2*:1 (February 1984), pp. 39-59. Originally presented at the *Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983.

[4] A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in the Emerald System," *OOPSLA'86 Conference Proceedings, 29* September - 2 October 1986, pp. 78-86. In *ACM SIGPLAN Notices 21*:11 (November 1986).

[5] A. P. Black, "Supporting Distributed Applications: Experience with Eden," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pp. 181-193. In *ACM Operating Systems Review 19*:5.

[6] C. M. Brown, R. J. Fowler, T. J. LeBlanc, M. L. Scott, M. Srinivas, and others, "DARPA Parallel Architecture Benchmark Study," BPR 13, Computer Science Department, University of Rochester, October 1986.

[7] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM TOCS 4*:2 (May 1986), pp. 110-129. Originally presented at the *Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985.

[8] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 129-140. In *ACM Operating Systems Review 17*:5.

[9] D. J. DeWitt, R. Finkel, and M. Solomon, "The Crystal Multicomputer: Design and Implementation Experience," *IEEE Transactions on Software Engineering SE-13*:8 (August 1987), pp. 953-966.

[10] R. Finkel and others, "Experience with Crystal, Charlotte, and LYNX," Computer Sciences Technical Reports #630, #649, and #673, University of Wisconsin – Madison, February, July, and November 1986.

[11] R. A. Finkel, M. L. Scott, Y. Artsy, and H.-Y. Chang, "Experience with Charlotte: Simplicity and Function in a Distributed Operating System," *IEEE Transactions on Software Engineering*, June 1989 (to appear). Extended abstract presented at the *IEEE Workshop on Design Principles for Experimental Distributed Systems*, Purdue University, 15-17 October 1986.

[12] J. P. Fishburn, "An Analysis of Speedup in Parallel Algorithms," Ph. D. thesis, Computer Sciences Technical Report #431, University of Wisconsin – Madison, May 1981.

[13] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley and Sons, New York, second edition, 1987.

[14] J. Kepecs and M. Solomon, "SODA: A Simplified Operating System for Distributed Applications," *ACM Operating Systems Review 19*:4 (October 1985), pp. 45-56. Originally presented at the *Third ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 27-29 August 1984.

[15] T. J. LeBlanc and R. P. Cook, "An Analysis of Language Models for High-Performance Communication in Local-Area Networks," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, 27-29 June 1983, pp. 65-72. In *ACM SIGPLAN Notices 18*:6.

[16] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS 5*:3 (July 1983), pp. 381-404.

[17] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, "Implementation of Argus," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 8-11 November 1987, pp. 111-122. In *ACM Operating Systems Review 21*:5.

[18] B. J. Nelson, "Remote Procedure Call," Ph. D. Thesis, Technical Report CMU-CS-81-119, Carnegie-Mellon University, 1981.

[19] M. L. Scott, "LYNX Reference Manual," BPR 7, Computer Science Department, University of Rochester, August 1986 (revised).

[20] M. L. Scott, "The Interface Between Distributed Operating System and High-Level Programming Language," *Proceedings of the 1986 International Conference on Parallel Processing*, 19-22 August 1986, pp. 242-249.

[21] M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering SE-13*:1 (January 1987), pp. 88-103.

[22] M. L. Scott and A. L. Cox, "An Empirical Study of Message-Passing Overhead," *Proceedings of the Seventh International Conference on Distributed Computing Systems*, 21-25 September 1987, pp. 536-543.

[23] M. L. Scott and R. A. Finkel, "A Simple Mechanism for Type Security Across Compilation Units," *IEEE Transactions on Software Engineering 14*:8 (August 1988), pp. 1238-1239.

[24] A. Z. Spector, "Performing Remote Operations Efficiently on a Local Computer Network," *CACM 25*:4 (April 1982), pp. 246-260.

[25] R. E. Strom and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, 27-29 June 1983, pp. 73-82. In *ACM SIGPLAN Notices 18*:6.

[26] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann, "A Structural View of the Cedar Programming Environment," *ACM TOPLAS 8*:4 (October 1986), pp. 419-490.

[27] W. F. Tichy, "Smart Recompilation," *ACM TOPLAS 8*:3 (July 1986), pp. 273-291. Relevant correspondence appears in Vol. 10, No. 4.

[28] United States Department of Defense, "Reference Manual for the Ada® Programming Language," (ANSI/MIL-STD 1815A-1983), 17 February 1983. Available as Lecture Notes in Computer Science #106, Springer-Verlag, New York, 1981.

[29] J. Welsh and A. Lister, "A Comparative Study of Task Communication in Ada," *Software — Practice and Experience 11* (1981), pp. 257-290.

[30] N. Wirth, "Modula: a Language for Modular Multiprogramming," *Software — Practice and Experience 7* (1977), pp. 3-35.